# High performance graph algorithms from parallel sparse matrices


**Viral Shah**
**University of California, Santa Barbara**

**John R. Gilbert, UCSB**

**Steven Reinhardt, SGI**

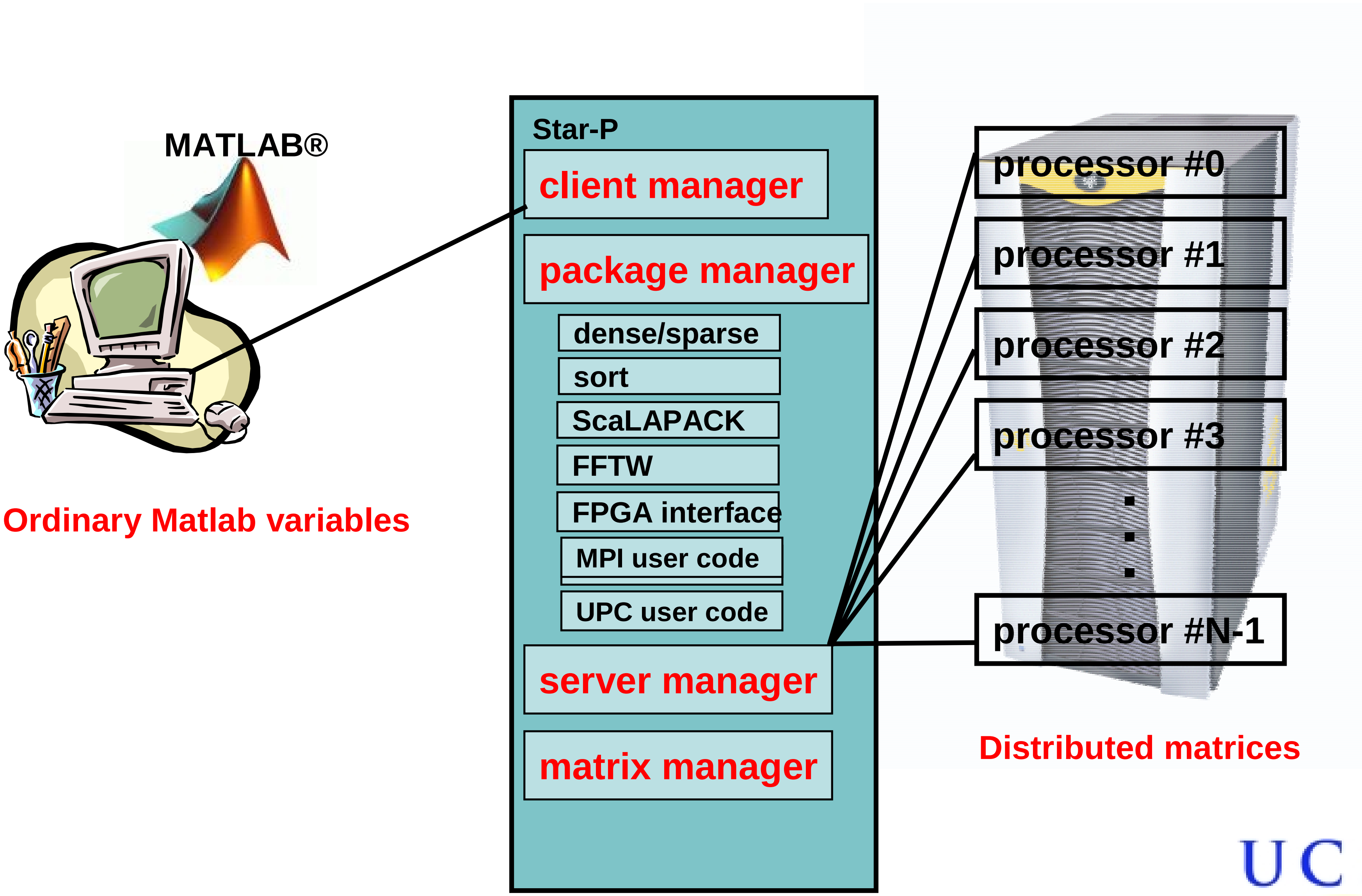


Support: DOE Office of Science, DARPA, SGI, ISC

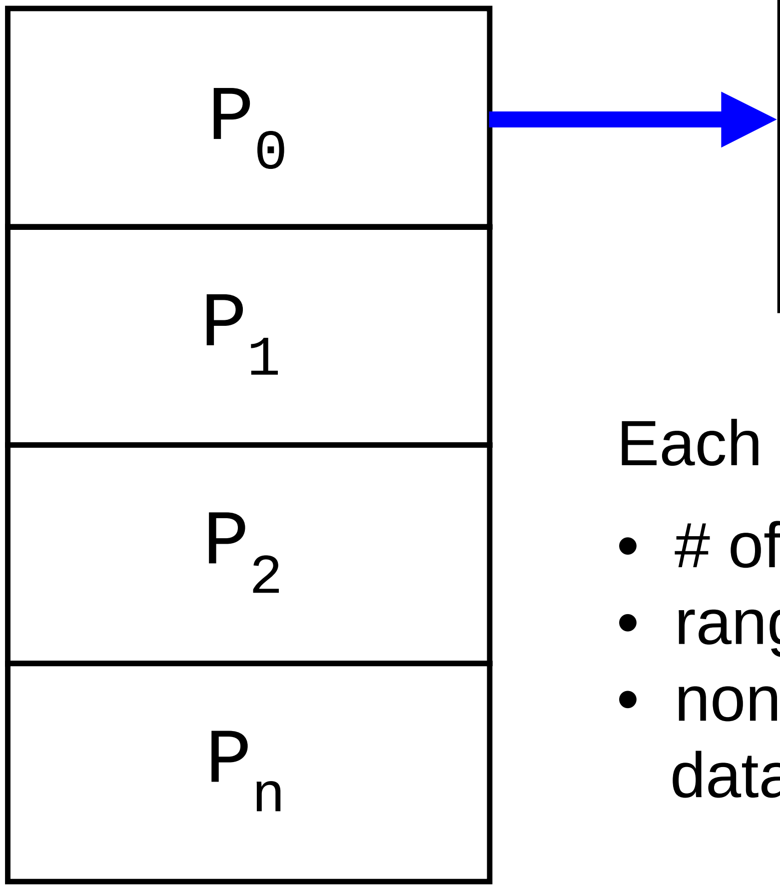
# Power method in Star-P

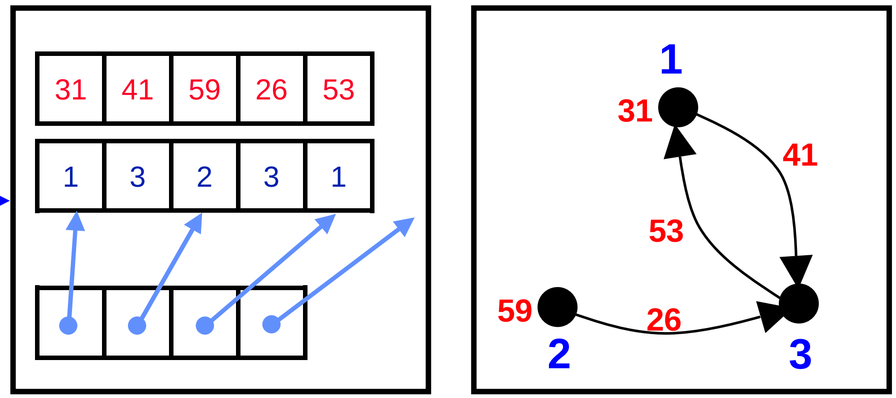
```
A = sprandn(4000*p, 4000, 0.1);
x = randn(4000*p, 1);
y = zeros(size(x));

while norm(x-y) / norm(x) > 1e-11
    y = x;
    x = A * x;
    x = x ./ norm(x);
end
```

# Star-P Architecture



# Distributed sparse array




Each processor stores:

- # of local nonzeros (# local edges)
- range of local rows (local vertices)
- nonzeros in a compressed row data structure (local edges)

# Sparse matrix operations

- A = sparse(i, j, Aij);
- [i j Aij] = find(A);
- Matrix operators: +, -, max, sum, & etc.
- matrix * vector, matrix * matrix
- Matrix indexing and concatenation

  A (1:3, [4 5 2]) = [ B(:, 7) C ] ;
- A \ b by direct methods (SuperLU_dist) and iterative methods
- Eigensolvers (PARPACK): eigs(), svds()



UCSB

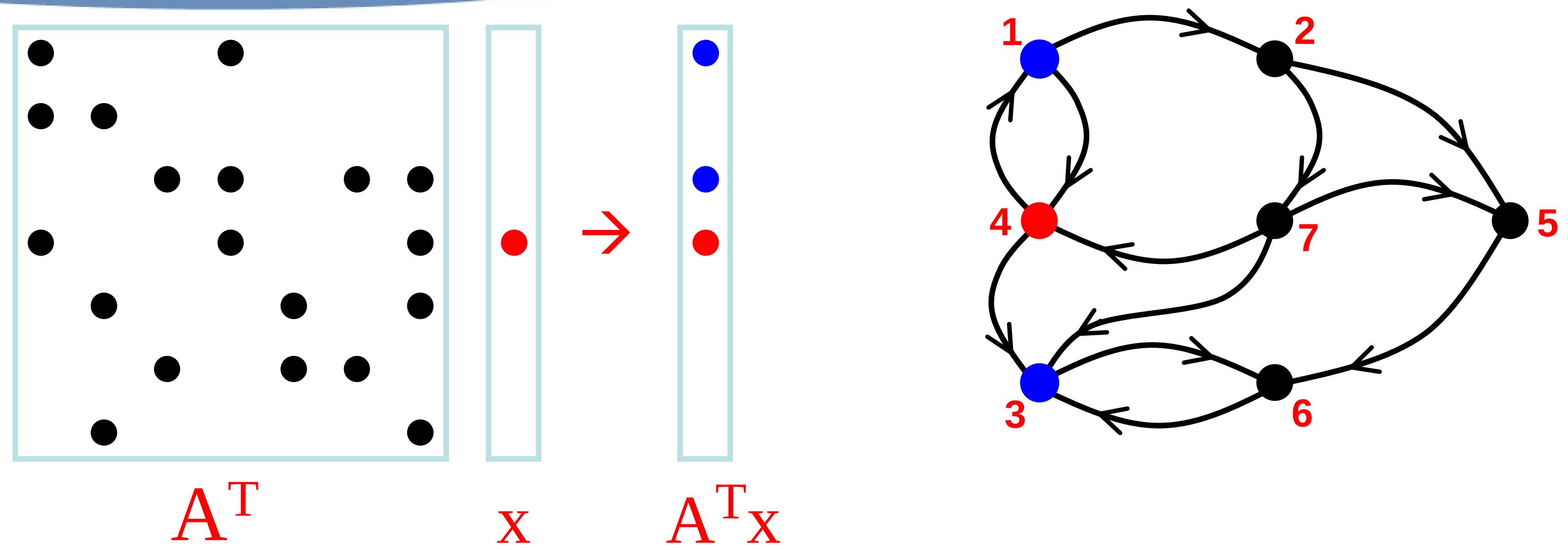
# Combinatorics in Star-P

- Represent a graph as a sparse adjacency matrix
- A sparse matrix language is a good start on primitives for computing with graphs
  - Random-access indexing: `A(i,j)`
  - Neighbor sequencing: `find (A(i,:))`
  - Sparse table construction: `sparse (I, J, V)`
  - Breadth-first search step : `A * v`



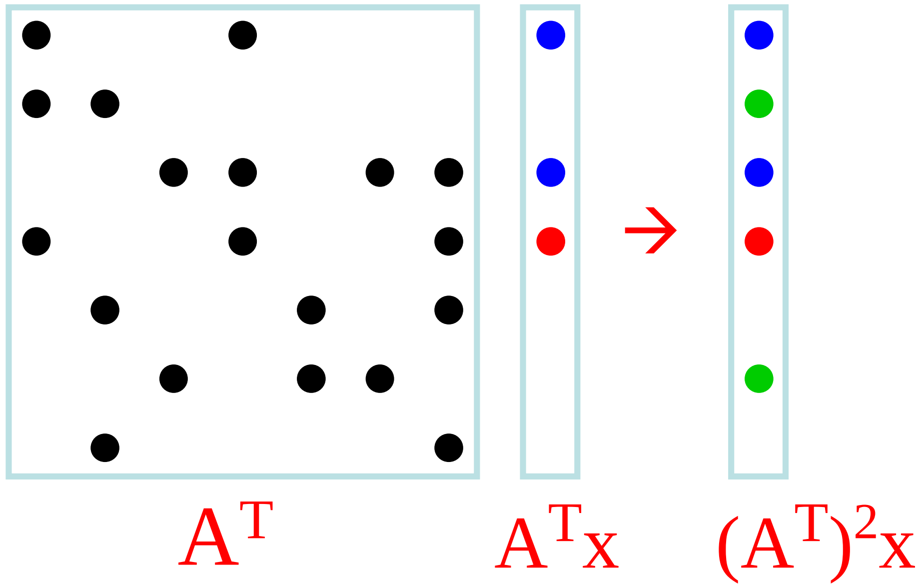
UCSB

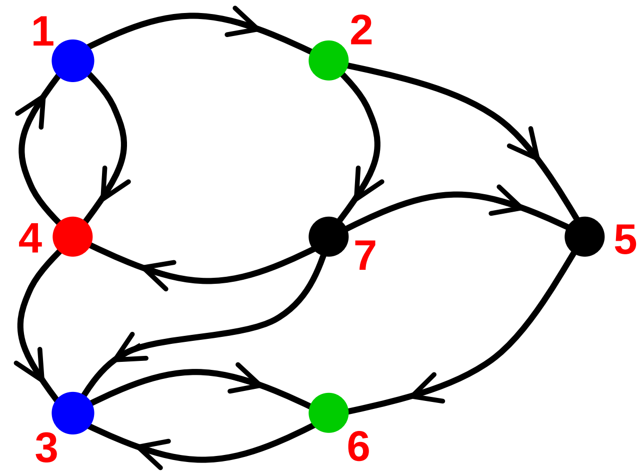
# Breadth-first search: sparse mat * vec




- Multiply by adjacency matrix → step to neighbor vertices
- Efficient implementation from sparse data structures

# Breadth-first search: sparse mat * vec

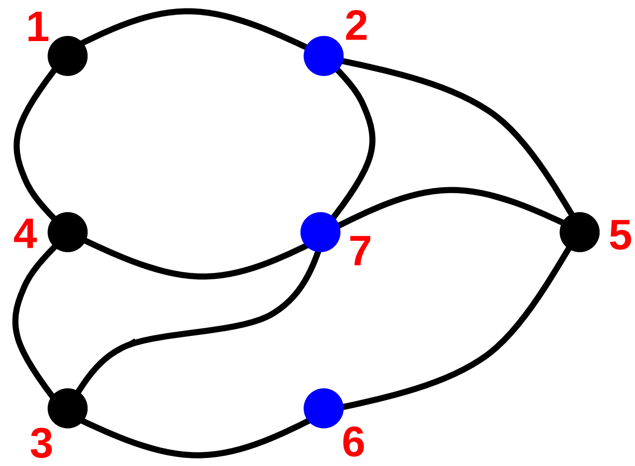



- Multiply by adjacency matrix → step to neighbor vertices
- Efficient implementation from sparse data structures

# Maximal Independent Set

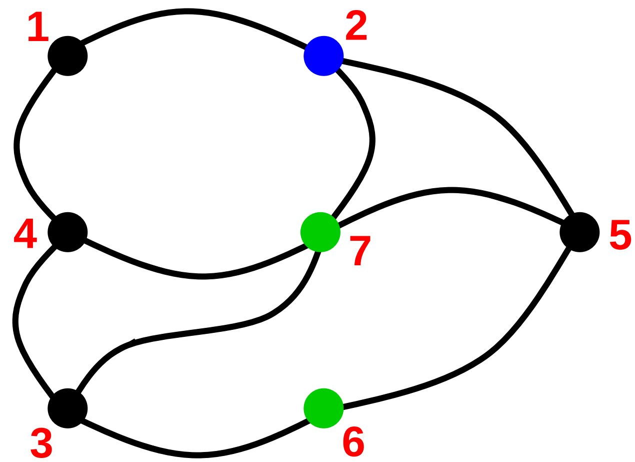
```
deg = sum(G, 2);
prob = 1 ./ (2 * deg);
select = rand (n, 1) < prob;

neigh = select & (G * select);
if ~isempty (neigh)
  % keep higher degree vertices
end
IS = [IS select];

neigh = neigh | (G * select);
remain = neigh == 0;
G = G(remain, remain);
```




Select a subset of graph vertices randomly as an initial guess of the independent set

# Maximal Independent Set

```
deg = sum(G, 2);
prob = 1 ./ (2 * deg);
select = rand (n, 1) < prob;

neigh = select & (G * select);
if ~isempty (neigh)
  % keep higher degree vertices
end
IS = [IS select];

neigh = neigh | (G * select);
remain = neigh == 0;
G = G(remain, remain);
```

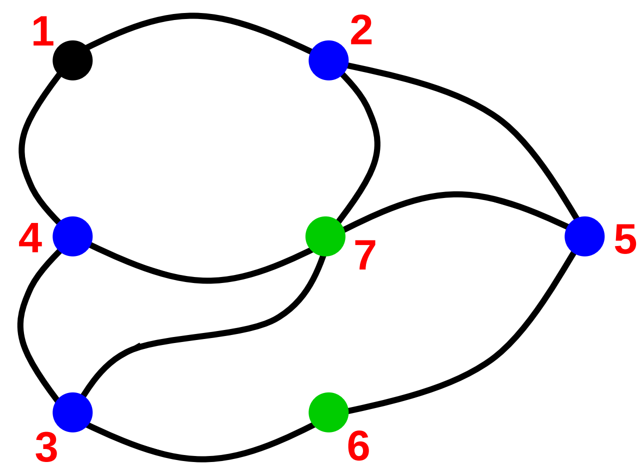



If neighbouring nodes are picked, keep the higher degree vertices.

Add selected vertices to the independent set.

# Maximal Independent Set

```
deg = sum(G, 2);
prob = 1 ./ (2 * deg);
select = rand (n, 1) < prob;

neigh = select & (G * select);
if ~isempty (neigh)
  % keep higher degree vertices
end
IS = [IS select];

neigh = neigh | (G * select);
remain = neigh == 0;
G = G(remain, remain);
```




Discard neighbours of the independent set.

Iterate the same process on the remaining subgraph.

# Connected components of a graph

- Sequential Matlab uses depth-first search (`dmperm`), which doesn't parallelize well
- Shiloach-Vishkin pointer-jumping algorithm:
  - repeat
    - Link every (super)vertex to a neighbor
    - Shrink each tree to a supervertex by pointer jumping
  - until no further change
- Hybrid SV / search method under construction
- Other possible graph kernels:
  - Shortest-path search (after Husbands, LBNL)
  - Bipartite matching (after Riedy, UCB)
  - Strongly connected components (after Pinar, LBNL)

# SSCA#2: “Graph Analysis”

QuickTime™ and a
Sorenson Video 3 decompressor
are needed to see this picture.

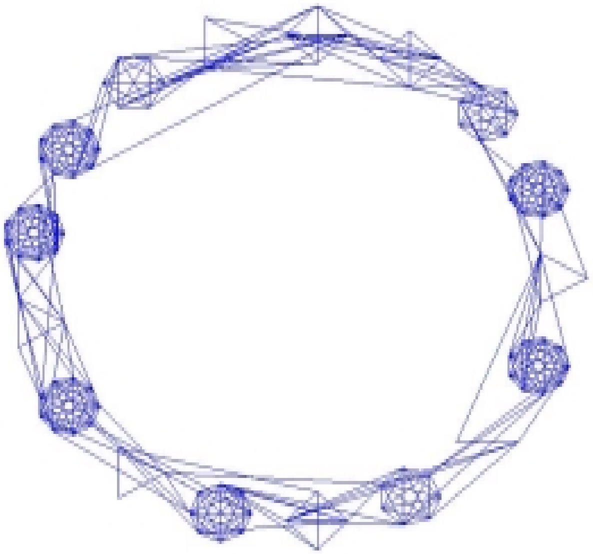
- Fine-grained, irregular data access
- Searching and clustering
- Goal is scaling to very large graphs
- Graphs specified by a scalable data generator

Four computational kernels:

- Kernel 1: Build graph data structure
- Kernel 2: Search by edge labels
- Kernel 3: Extract subgraphs
- Kernel 4: Partition into clusters

UCSB

# SSCA#2: Graph Statistics

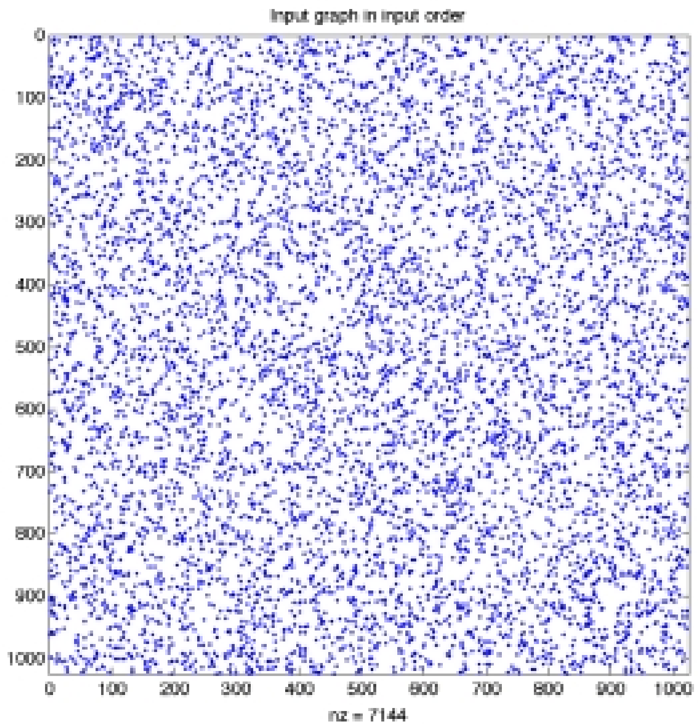


- Scalable data generator (Spec 1.1)
- Input data is edge triples **<** i, j, weight(i,j) **>**
- Many tight clusters, loosely interconnected
- Vertex and edge orders permuted randomly

| Scale | #Vertices | #Cliques | #Edges Directed | #Edges Undirected |
|---|---|---|---|---|
| 10 | 1,024 | 186 | 13,212 | 3,670 |
| 15 | 32,768 | 2,020 | 1,238,815 | 344,116 |
| 20 | 1,048,576 | 20,643 | 126,188,649 | 35,052,403 |
| 25 | 33,554,432 | 207,082 | 12,951,350,000 | 3,597,598,000 |
| 30 | 1,073,741,824 | 2,096,264 | 1,317,613,000,000 | 366,003,600,000 |

UCSB

# Concise SSCA#2 in Star-P




**Kernel 1:** Construct graph data structures

- Graphs are dsparse matrices, created by sparse( )

# Kernels 2 and 3

## Kernel 2: Search by edge labels

- About 12 lines of executable Matlab or Star-P
- Essentially uses find()

## Kernel 3: Extract subgraphs

- Returns subgraphs consisting of vertices and edges within fixed distance of given starting vertices
- Sparse matrix-matrix multiplication for several simultaneous breadth-first searches
- About 25 lines of executable Matlab or Star-P

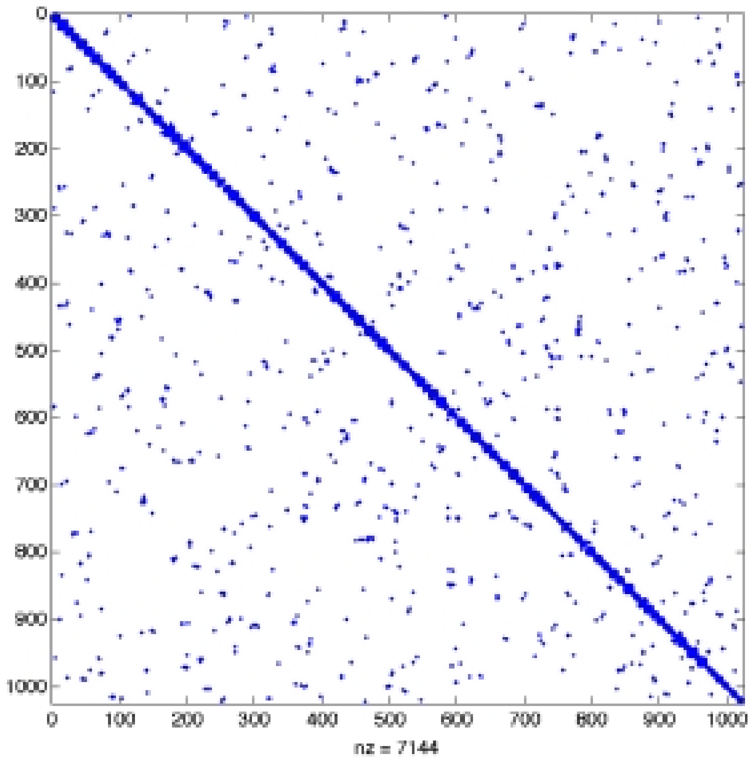


UCSB

# Kernel 4: Vertex clustering

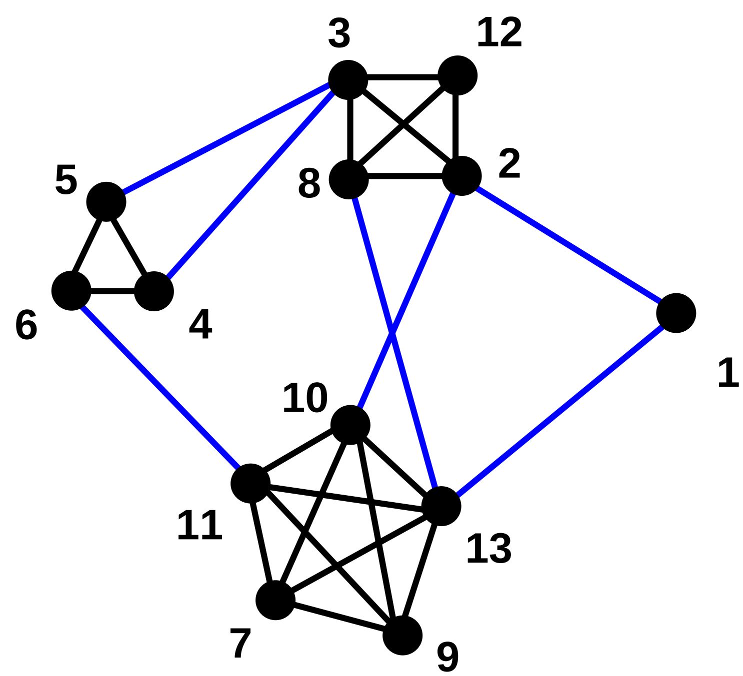
- Grow local clusters from many seeds in parallel
- Breadth-first search by sparse matrix * matrix

```
% Grow each seed to vertices
%    reached by at least k
%    paths of length 1 or 2

 C = sparse(seeds,1:ns,1,n,ns);
 C = A * C;
 C = C + A * C;
 C = C >= k;
```



# Kernel 4: Peer pressure

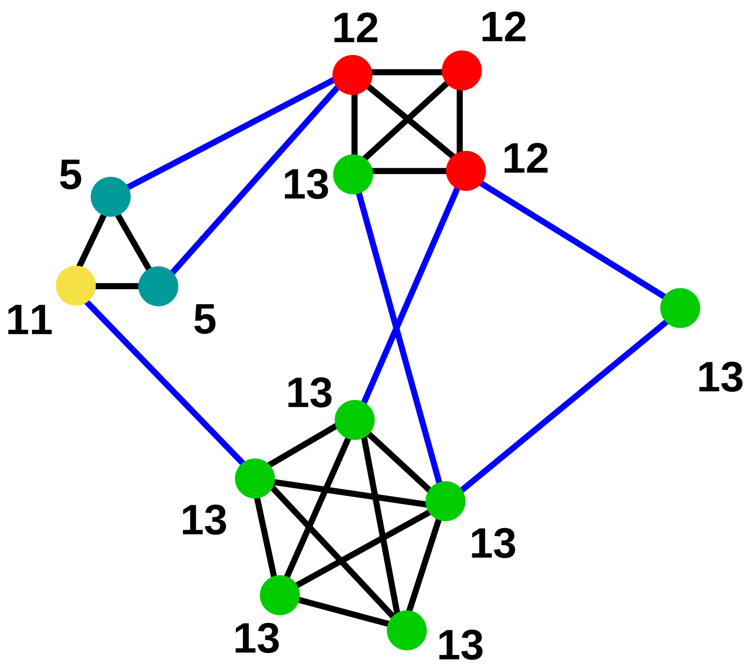
```
Steps in a peer pressure algorithm:
   1. Vote for a leader
   2. Collect neighbour votes
   3. Vote for a new leader
      (based on neighbour votes)
```




- Quality of clustering depends on the choice of algorithms used for the different steps above.
- The set of possible leaders should be small. MIS is a good choice. For SSCA#2, max works equally well.
- Neighbour votes maybe combined using different weights.
- All versions of kernel4 are about 25 lines of code.

# Kernel 4: Peer pressure

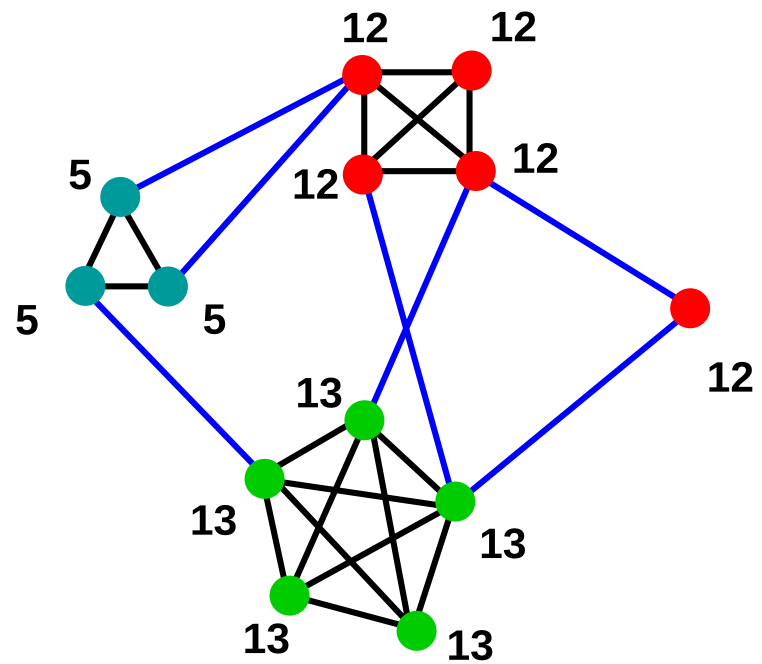
```
[ign leader] = max (G, [], 2);

S = G *
  sparse(1:n,leader,1,n,n);

[ign leader] = max (S, [], 2);
```




- Every vertex votes for its highest numbered neighbour as its leader - No communication is required
- The size of the leader set is approximately the number of clusters - which is small relative to the number of nodes
- Discovers original graph structure right away

# Kernel 4: Peer pressure

```
[ign leader] = max (G, [], 2);

S = G *
  sparse(1:n,leader,1,n,n);

[ign leader] = max (S, [], 2);
```




- Matrix multiplication gathers neighbour votes
- Every nonzero in each row corresponds to a leader - Its value denotes the number of neighbour votes for that leader
- >95% of the original graph structure is recovered at this point
- Very small clusters may attach themselves to nearby clusters

# Scaling up

Recent runs of cSSCA#2 on SGI Altix (up to 128 processors):

- Have run the entire benchmark on graphs with $2^{26}$ = 67 million vertices, 890 million directed edges, 247 million undirected edges - (ver 0.9 of the spec)
- Benchmarking in progress for ver 1.1 of the spec
- Have built graphs with 400 million vertices and 4 billion edges
- Timings scale well – for large graphs,
  - 2x problem size → 2x time
  - 2x problem size & 2x processors → same time

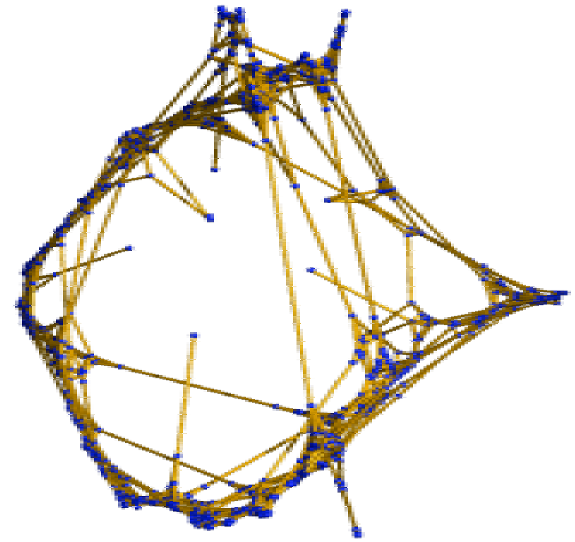


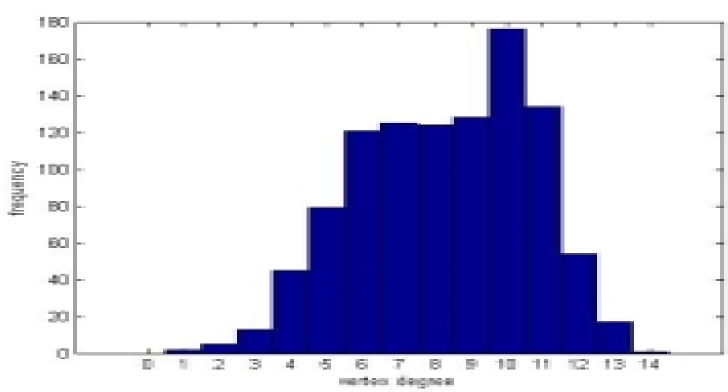
UCSB

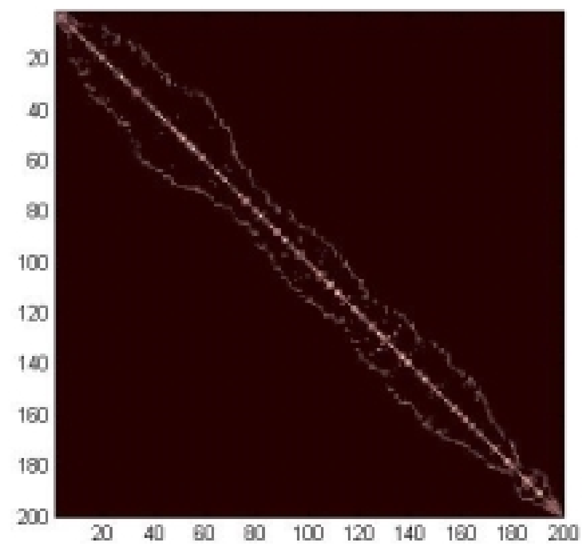
# Toolbox for Graph Analysis and Pattern Discovery

**Layer 1: Graph Theoretic Tools**

- Graph operations
- Graph generators
- Graph partitioning and clustering
- Graph theoretic preconditioners
- Visualization and graphics
- Scan and combining operations
- Utilities





# Thank You

Questions.

# Toolbox Status

- **Graph algorithms**: independent sets, connected components, strongly connected components, shortest paths, bipartite matching, graph coloring, spanning trees
- **Graph partitioning and mesh generation**: Simple 2d and 3d mesh generators, stencil operators, spectral partitioners, geometric partitioners, multilevel partitioners (ParMETIS hookup)
- **Solution of linear systems**: Preconditioned iterative methods, support graph preconditioners, algebraic multigrid, sparse approximate inverse preconditioners



UCSB

# Extra Slides